UCRL-CONF-212857

Computer Language Choices in Arms Control and Nonproliferation Regimes

G. K. White

June 10, 2005

46th Annual Meeting of the Institute of Nuclear Materials Management
Phoenix, AZ, United States
July 10, 2005 through July 14, 2005

Computer Language Choices in Arms Control and Nonproliferation Regimes

Greg White
Lawrence Livermore National Laboratory
June 2005

Abstract
The U.S. and Russian Federation continue to make substantive progress in the arms control and nonproliferation transparency regimes. We are moving toward an implementation choice for creating radiation measurement systems that are transparent in both their design and in their implementation. In particular, the choice of a programming language to write software for such regimes can decrease or significantly increase the costs of authentication. In this paper, we compare procedural languages with object-oriented languages. In particular, we examine the C and C++ languages; we compare language features, code generation, implementation details, and executable size and demonstrate how these attributes aid or hinder authentication and backdoor threats. We show that programs in lower level, procedural languages are more easily authenticated than are object-oriented ones. Potential tools and methods for authentication are covered. Possible mitigations are suggested for using object-oriented programming languages.

1. Introduction to Authentication

As we continue to make progress towards the development monitoring systems for nuclear material, two important goals must be observed: protection of the host country's sensitive information and the assurance to the monitoring party that the nuclear material is what the host country has declared it to be. These goals are met by certification in the host country and authentication in the monitoring party. During both certification and authentication, each party will need to understand all of the operating parameters of all hardware and software in the deployed system. This paper will concentrate on authentication, but similar principles apply to certification as well.

Authentication is the process of gaining assurance that a system is performing robustly and precisely as it is intended. The simpler you make the system, the easier it is to authenticate. It is important to limit functionality to only what is needed to satisfy the requirements of the task. Each design decision makes authentication easier, or harder. For example, a design with MSDOS (which required a 4.77 MHz processor and ran on a single floppy disk) would be easier to authenticate than a Windows XP installation (which requires a 300 MHz Pentium Processor and 1.5GB of hard disk).[1] Simpler hardware, expressed in the number of gates, chips, or boards, is easier to authenticate than more complex hardware. The same can be said for software.

In my previous INMM paper[2], I discussed a hypothetical perfect system for authentication, with transparent (to both parties) hardware and software development, and advocated "open source" hardware and software solutions. In this paper, I will advocate software language choice which eases authentication costs.

Other industries have a similar need for authentication. Computers which perform electronic voting[3] and gambling are two examples.

2. Authenicating C++

C++ is an object oriented programming language created by Bjarne Stroustrup at Bell Labs between 1983 and 1985.[4] Although a perfectly good language for many tasks, it severely complicates authentication over simpler procedural languages, such as C. The similarity in the names of the two languages leads to the notion that C++ is just a minor enhancement to the C language. The huge increases in compiler complexity and code generation in C++ is hidden from the user.

One of the primary tenets of object oriented programming (i.e. C++) is the ability to create hierarchies of objects and a series of ever more complex abstractions. A simple example of an object hierarchy would be: living things, mammals, humans, employees, computer nerds. C++ is harder to follow by source code inspection, since each object and method (i.e. function call) can be overridden (i.e. do something completely different) at each level of the object hierarchy.

The C linker is very simple; it binds the actual address of external functions in an object file. The C++ linker is actually an extension of the compiler. When you "link" a C++ code you may get *n!* invocations of the compiler to instantiate all possible classes and special methods.[5] This is not a simple case of one source file equals one object file. This leads to a further, nearly humanly unpredictable, complexity and obscurity.

C++ performs more tasks at runtime that can hinder authentication. It requires a larger, more complex runtime system to handle name binding, dynamic typecasting, memory allocation, etc. C++ (through its virtual function facility) requires a runtime system to bind actual functions to a particular virtual method. Tasks which happen at runtime are not viewable by static inspection of source and binary code. Even an outwardly simple thing like the allocation of space for an object can be complicated. An object can override the new method (which creates the object) so that side effects (not visible at the point of invocation) can occur. This is, in fact, how specialized, high-speed allocators are built for the Standard Template Library (STL, a collection of standard functions and methods for C++) and small-objects.

C++ generated object or assembly code is much harder to verify than its C counterpart. Non-optimized machine code (derived from C source code) can (in general) be decompiled back to something that resembles the original C source code at least insofar as comparing it to the original source. C++ goes through more intermediate steps in its translation, so the one-to-one mapping of statement to generated code is obscured. Consider the output of a simple C++ to C compiler like KAI[6] or cfront[7]. The generated C code bears little resemblance to the original C++ code even when compared side to side. The assembly code is much further still from the original.

It is harder to perform authentication on C++ source code because of compiler intervention to build complex structures, code motion to support inlining, and templatization in which the compiler crafts code on the behalf of the programmer. Additionally, the same problem of following dynamic actions caused by executing virtual functions means that it is difficult to trace affects without access and inspection of the full code base. For all practical purposes, C++ generated assembly code is not humanly inspectable. This leaves only inspection of the source, clean-room verification of compiling/linking and functional testing as the only viable tools for authentication.

C++ executables are (in general) larger than the equivalent C executable. This is due to the large number of small functions, the necessary runtime systems to handle dynamic binding, and an Input/Output system that is vastly more general and larger. The simple function "cout" (which writes characters to the standard output device) may be between 10 and 15 levels of deep in the object hierarchy of the iostream C++ library. In a degenerate case, a simple program which types "Hello World" on the screen, the C version of the executable was 2,000 bytes, while the C++ version was 206,000 bytes. While some of this will remain a constant size difference to any code, some of the code expansion will be proportional to the size of the source code.

Even the way that programmers tend to write code in a particular language can help or hinder authentication. C++ tends to be written with hundreds of small functions and methods to build up complex abstractions. While this aids programming, it hinders authentication.

C++ is many times harder and costlier to authenticate by inspection. C++ is so much more complex than C that automated tools are practically a requirement for "proving" correctness of C++ code. Such a tool would need to be created or customized, as it does not currently exist. This leaves only inspection of the source code, clean-room verification of compiling/linking, and functional testing as the only viable tools for authentication.

C++ is at least twice as costly (in time and money) to authenticate (by inspection) as C. Code inspection will be by guided walkthroughs of the code. Unlike C source code, each release of source code set in C++ must be viewed and inspected in its entirety [i.e. if you change one file you must reinspect all files], because of the possibility of subtle changes in the object hierarchy (i.e. easy-to-obscure, hard-to-handle complexity explosion). Object oriented programming methods are harder to follow by inspection, because they can induce side effects. Object oriented programming allows for code overlay which can extend and enhance capabilities. Inspection of subsequent version of C source code will concentrate primarily on the differences between the current and previous versions.

Since the complexity of code is a function of the number of function points (instead of Source Lines of Code, aka SLOC). There is 3-4 times the number of function points in C++ versus C because the language encourages localization of effect through small methods and attribute access functions.

Since there is a straightforward transliteration between C and its underlying assembly, it is harder to obscure effect in a C program. In C++, where any operation can be overloaded and it is less clear from the source where an object is a value, pointer, or reference, the translation of any operation into associated assembly requires access and understanding of the full source.

For example, the expression $A+B$ in the C language can only refer to adding some sort of integer, float, or pointer. No side effect is possible. One would expect to find some kind of assembly instruction implementing the addition. In C++, $A+B$ can be anything. The statement cannot be understood without fully inspecting the implementation of the class definitions for A and B (and if A and B can be pointers or references, all derived classes!). There are no automatic source code verification systems for C++, where there are some for C.

3. Mitigating Language Choice

A small percent of the additional authentication costs of choosing C++ can be reduced by imposing limitations on the programmer. These limitations remove some of the language features that hinder authentication the most. Here are some examples:

- Don't use Virtual methods.[8]
- Restrict the use of overloading of functions to help reduce name confusion
- Don't use default arguments in functions
- Do not use overloaded operator new() except in system and STL headers
- Do not allow dynamic casting of pointers in C++
- Discourage the use of templatization outside the STL.

4. Additional limitations

Even with a procedural language such as C, some additional programmer limitations can ease authentication:

- Encourage the use of system calls which do not allow for buffer overflows (gets vs fgets, strcpy vs strncpy)
- Turn off compiler optimizations
- Use only static loading, no dynamic loading of object files or Dynamic Loaded Libraries (DLL)
- Use a malloc() library that detects buffer underflow and overflow [e,.g. malloc_debug]
- Consider self-check of dynamic executable's MD5/SHA checksum
- Dynamic casting of C pointers should be discouraged
- Encourage the liberal use of assertions [e.g. design-by-contract] to verify that pointers are non-null, type values are consistent, etc.
- Be cautious with the use of asynchronous signal handlers and the "volatile" data type designation

---

[1] http://www.microsoft.com/windowsxp/home/evaluation/sysreqs.mspx

[2] White, G., Increasing Inspectability of Hardware and Software for Arms Control and Nonproliferation Regimes, Proceedings of the INMM 2001 Annual Meeting, Indian Wells, California

[3] As an aside, a genius co-worker of mine stated, "If I wanted to rig an election with an electronic voting machine, and I could choose any computer language to write my hide my deception in, I'd do it in C++"

[4] http://www.research.att.com/~bs/C++.html

[5] N factorial because each code object can request a new pass over all other code objects to build required templated classes and specialized methods

[6] Bought by Intel in 2000 and currently being phased out as a product.

[7] The original C++ compiler developed by Bjarne Stroustrup.

[8] Although this may seem to preclude all inheritance since C++ only allows inheritance of classes with virtual methods, a small exception can allow one "useless" method of the form "virtual void useless(void) {}" to allow inheritance where required.

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.